computer organization and structure course design

CPU system design

-------(CPU)

Southeast University

College of Information Science and Engineering

04021521 Wang Mengyun 04021414 Guo Rong

2023.5.25

1. The purpose of the experiment

To design and verify a simple CPU (Central Processing Unit) based on microprograms, the CPU consists of a basic instruction set in which each instruction consists of a set of microinstructions, which corresponds to a microprogram. In order to verify the correctness of the CPU design, an assembly program is written using some of the instructions in the instruction set, and the correctness of the CPU design is verified by comparing the results of theoretical calculations with the results of CPU operation.

2. experimental requirements

- CPU functions.
  - （1） Fetch instructions: the CPU must read instructions from memory.
  - （2） Translate instructions: instructions need to be interpreted as actions to be performed.
  - （3） Fetch data: the execution of an instruction requires data to be fetched from memory or an input/output port.
  - （4） Processing of data: the execution of instructions requires arithmetic or logical operations on data.
  - （5） Write Data:The result of instruction execution needs to be written to memory or input/output ports.
- CPU Tasking.
  - （1） Instruction set design.
  - （2） Memory and internal register design.
  - （3） Arithmetic logic unit design.
  - （4） Microprogrammed programming control unit design.

3. CPU architecture

According to the Design Requirements document and the textbook Computer Organization and Architecture-designing for performance in CPU

The simple CPU architecture realized in this experiment, considering the actual control signal distribution, is shown in the following figure.

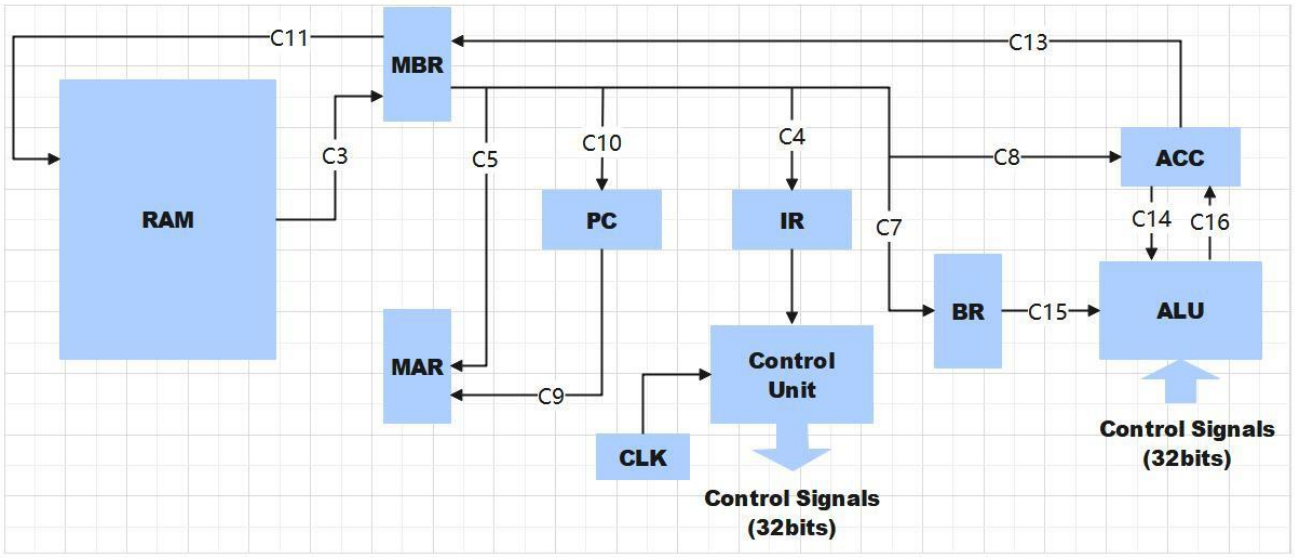


Figure 1. CPU data paths and control signals

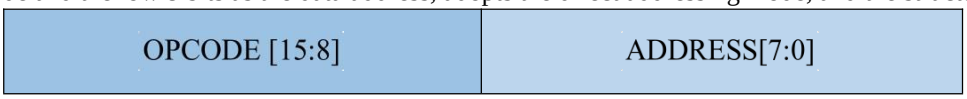
CPU operation realization flow.

When the CPU executes an instruction, the address of the instruction is stored in the PC. First of all, PC transfers the stored address to MAR, and according to the address, it takes out the instruction data from Memory, including the opcode and operand, totaling 16 bits, and puts it into MBR. After that, the IR reads the upper 8 bits of the instruction data stored in the MBR, i.e., the opcode, and sends it to the CU to determine the type of the current instruction. After determining the type of the instruction, the CU determines the start address of the corresponding microprogram, and executes the microprogram operations sequentially by outputting control signals until the end of the microprogram. In this way, the CPU completes the corresponding operation of the instruction.

4. instruction set design

- Command Format.

  CPU adopts single-address instruction format, i.e., an instruction contains two parts, the opcode part that defines the function of the instruction and the address part that defines the location of the operand. 16-bit instruction, with the high 8 bits as the opcode and the low 8 bits as the data address, adopts the direct addressing mode, and the structure is as follows.



- Instruction set.

Command	opcodes	Operation
STORE X	0000 0001	ACC→[X]
LOAD X	0000 0010	[X]→ACC
ADD X	0000 0011	ACC+[X]→ACC
SUB X	0000 0100	ACC−[X]→ACC
JMPGEZ X	0000 0101	If ACC≥0 then X→PC else PC+1→PC
JMP X	0000 0110	X→PC
HALT	0000 0111	Suspending the compilation of the program
MPY X	0000 1000	ACC×[X]→ACC, MR
AND X	0000 1010	ACC and [X]→ACC
OR X	0000 1011	ACC or [X]→ACC
NOT X	0000 1100	NOT [X]→ACC
SHR	0000 1101	ACC Shift right one bit, logical shift, left complement 0
SHL	0000 1110	ACC Shift left one bit, logical shift, right complement 0
SAR	0000 1111	ACC 右移一位，算术移位，左边补符号位
SAL	0001 0000	ACC Shift left by one, arithmetic shift, right by 0.

Table 1: Instruction set and related opcodes In the above table, [X] indicates the content stored in memory at address X. The following table shows the instruction set and related opcodes.

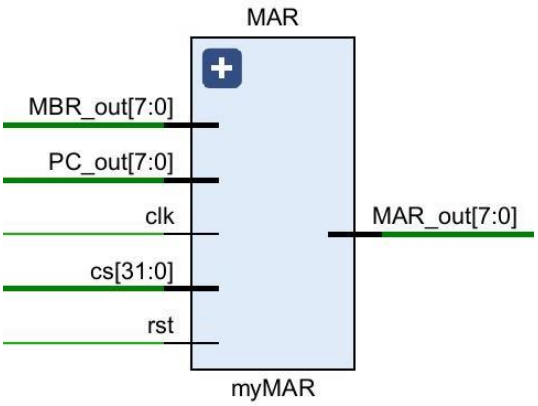
5. internal registers and memory design

- ***M***

A stores the memory address of the upcoming read/write operation, in this design *A* is 8bits, can address 256 memory data.

MAR inputs: Clock signal (1 bit), control signal (32 bits), PC input address (8 bits), MBR input address (8 bits).
MAR output:Address of MAR output (8bit).

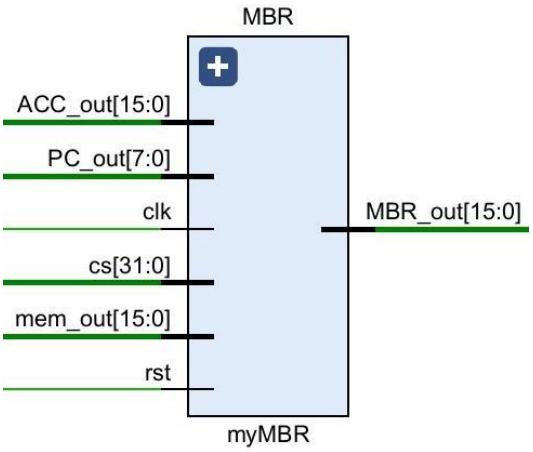
- Input and output port diagrams.



-

M holds the data that is about to be written to memory or just read out from memory, in this design, *M* is 16bits.
Inputs to MBR: Clock signal (1 bit), control signal (32 bits), input to ACC (16 bits), data read from RAM (16 bits), input to PC (16 bits).
Output of MBR:Data written to memory by MBR (16bit).

- Input and output port diagrams



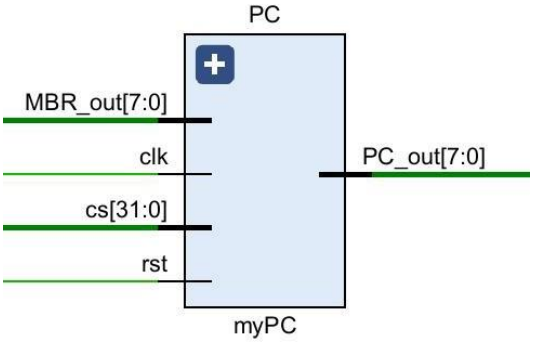
- ***P***

P holds the address of the next executed instruction, in this design, *P* is 8bits.

Inputs to PC: Clock signal (1bit), control signal (32bit), and command input from MBR (8bit).

The output of PC is: Output address (8bit).

- Input and output port diagrams



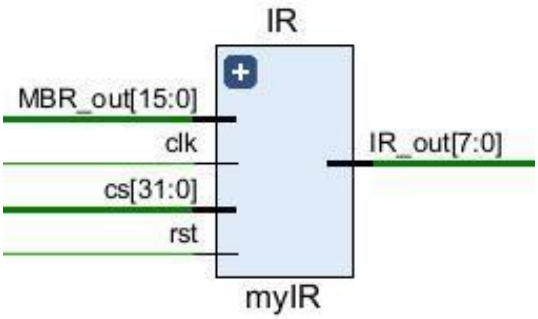
-

A stores the opcode of the current instruction, in this design, *A* is 8bits.

The inputs to the IR are: clock signal, control signal (32bit), and command input from MBR.

The output of IR is: Output command (8bit).

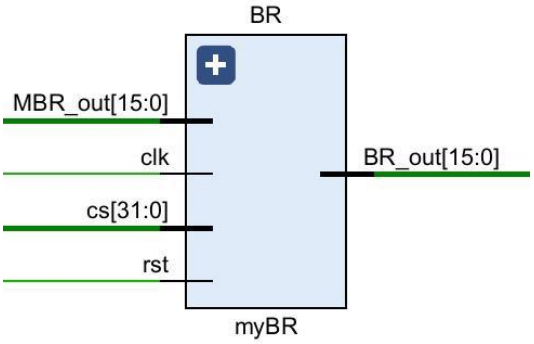
- Input and output port diagrams



-

is an input to *A* containing an operand, in this design, of 16bits.

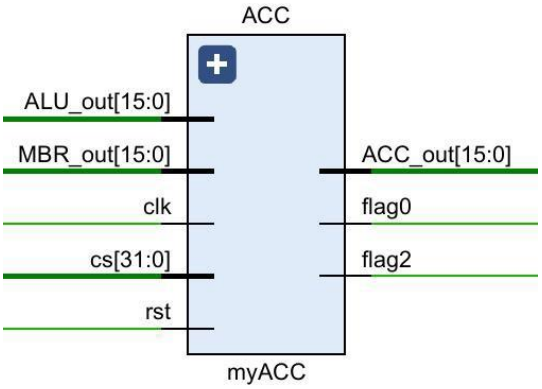
- Input and output port diagrams



-

P is an input to the ALU and is used as the result output of the ALU, in this design, the ACC is 16bits.

- Input and output port diagrams



(Note: flag0 is the result positive or negative flag, flag2 is the end of processing flag)

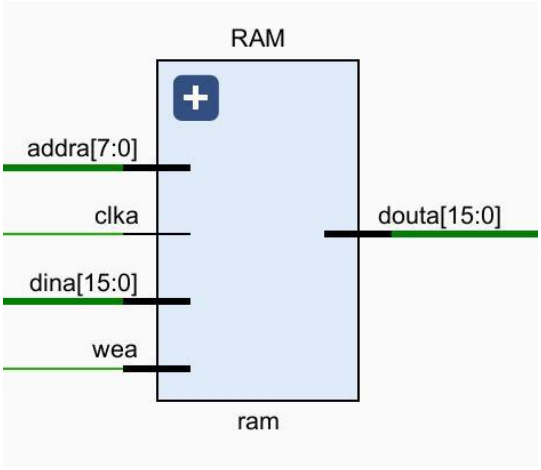
- ***M***

In this design, the size of the program is 256×16 bits.

The inputs to the RAM are: clock signal (1 bit), read/write control signal (1 bit), data to be written to the RAM (16 bits), and the address of the storage cell to be operated (8 bits).

The output of RAM is: data read out to MBR (16bit).

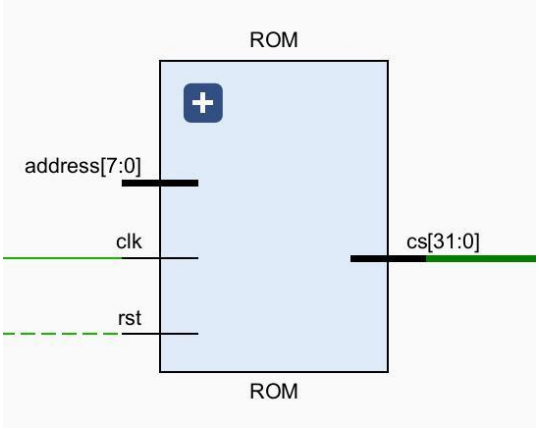
- Input and output port diagrams



- ***R***

In this design, the control signals are 32bits and the size is 256×32bits, which are stored in the control signals for the micro-operations corresponding to each instruction.

- Input and output port diagrams



6. ALU Design

- Function Description

ALU is a module in a computer that performs arithmetic and logical operations. The functions to be accomplished by the arithmetic logic unit in this design are shown in the following table.

Operations	Explanation
ADD	(ACC)<=(ACC)+(BR)
SUB	(ACC)<=(ACC)-(BR)
MPY	(ACC)<=(ACC)×(BR)
AND	(ACC)<=(ACC) and (BR)
OR	(ACC)<=(ACC) or (BR)
NOT	(ACC)<=not (ACC)
SHR	(ACC)<= (ACC)>>1
SHL	(ACC)<= (ACC)<<1

SAR	(ACC)<= (ACC)>>>1
SAL	(ACC)<= (ACC)<<<1

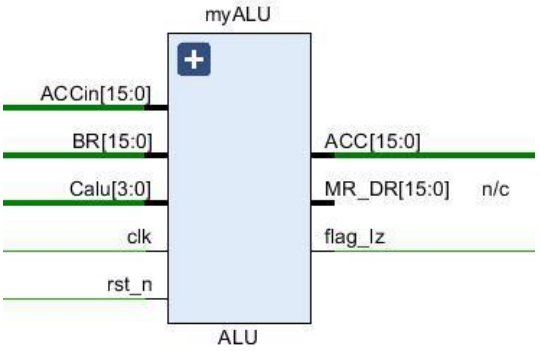
Table 2. Algorithms that can be handled by the ALU

Complementary arithmetic is
used in this design.

The following table shows the three registers that are closely related to the ALU and their related information.

Name	Bitmaster	Function
BR	16bit	The input to the ALU, containing an operand
ACC	16bit	The accumulator, as an operand of the ALU, is usually used to store the result
MR	16bit	Prepare for multiplication operation, store multiplication result in high 16bit

- Input and output port diagrams



7. Control Unit design

According to the CPU structure shown in Figure 1, each instruction can be decomposed into multiple micro-operations. Microprogram has a series of micro-instructions, these micro-instructions stored in the CU (Control Unit, Control Unit) control storage, each micro-instruction contains one or more micro-operations, micro-operations by the control signals to achieve the Control Unit only need to generate the corresponding instruction control signals to complete the execution of the instructions. CAR (Control Address Register, Control Address Register) is used to store the address of a storage unit in the control memory to be read, CBR (Control Buffer Register, Control Buffer Register) is used to temporarily store the 32bit control signals read from the control memory, and the next clock will be the control signal output.

When the program starts to execute, CU firstly executes the instruction fetch operation, through a series of micro-instruction and micro-operation, it fetches the first instruction into IR, and updates PC at the same time, after that, CU determines the specific instruction to be executed and whether it is necessary to fetch instruction according to the current control signals, and then executes the next instruction, outputs a group of control signals step by step in the order, and so on until it encounters the HALT (Stop Instruction), which completes the whole program execution. The execution of the whole program is completed.

- Definition of control signals and corresponding micro-operations.

control signals	Micromanipulation
C0	CAR<=CAR+1
C1	CAR<=IR(opcode)
C2	CAR<=0
C3	MBR<=memory
C4	IR<=MBR[15:8]
C5	MAR<=MBR[7:0]
C6	PC<=PC+1
C7	BR<=MBR
C8	ACC<=MBR

C9	MAR<=PC
C10	PC<=MBR[7:0]

C11	mem<=MBR
C12	ACC<=0
C13	MBR<=ACC
C14	ALU<=ACC
C15	ALU<=BR
C16	ACC<=ALU
C17	CAR<=CAR+1+flag(1)
C18	CAR<=CAR+flag(3)
C19	empty
C20	ALU<=ACC+BR
C21	ALU<=ACC-BR
C22	ALU<=ACC and BR
C23	ALU<=ACC or BR
C24	ALU<=not ACC
C25	ALU<=logicshift(ACC)to left 1bit
C26	ALU<=logicshift(ACC)to right 1 bit
C27	ALU<=ACC*BR
C28	ALU<= ACC/BR DR<=ACC%BR
C29	ALU<= arthshift(ACC)to left 1bit
C30	ALU<=arthshift(ACC)to right 1bit
C31	empty

Table 3. Control signal definitions

- Micro-operation and control signals corresponding to commands.

	CAR address		control signals
fetch	0	MAR<=PC,CAR<=CAR+1	C9，C0
	1	MBR<=memory,CAR<=CAR+1	C3, C0
	2	IR<=MBR[15:8],CAR<=CAR+1	C4, C0
	3	CAR<=(opcode)	C1
Load(02)	4	MAR<=MBR[7:0],PC<=PC+1,CAR<=CAR+1	C5，C6，C0
	5	MBR<=memory,CAR<=CAR+1	C3, C0
	6	ACC<=MBR,CAR<=0	C8，C2
STORE(01)	7	MAR<=MBR[7:0],PC<=PC+1,CAR<=CAR+1	C5，C6，C0
	8	MBR<=ACC,CAR<=CAR+1	C13, C0
	9	memory<=MBR,CAR<=0	C11, C2
ADD(03)	10	MAR<=MBR[7:0],PC<=PC+1,CAR<=CAR+1	C5，C6，C0
	11	MBR<=memory,CAR<=CAR+1	C3, C0
	12	BR<=MBR,CAR<=CAR+1	C7,C0
	13	ALU<=BR,ALU<=ACC,CAR<=CAR+1	C15,C14,C0
	14	ALU<=ACC+BR,CAR<=CAR+1	C20,C0
	15	ACC<=ALU,CAR<=0	C16,C2
SUB(04)	16	MAR<=MBR[7:0],PC<=PC+1,CAR<=CAR+1	C5，C6，C0
	17	MBR<=memory,CAR<=CAR+1	C3, C0
	18	BR<=MBR,CAR<=CAR+1	C7,C0
	19	ALU<=BR,ALU<=ACC,CAR<=CAR+1	C15,C14,C0
	20	ALU<=ACC-BR,CAR<=CAR+1	C21,C0
	21	ACC<=ALU,CAR<=0	C16,C2

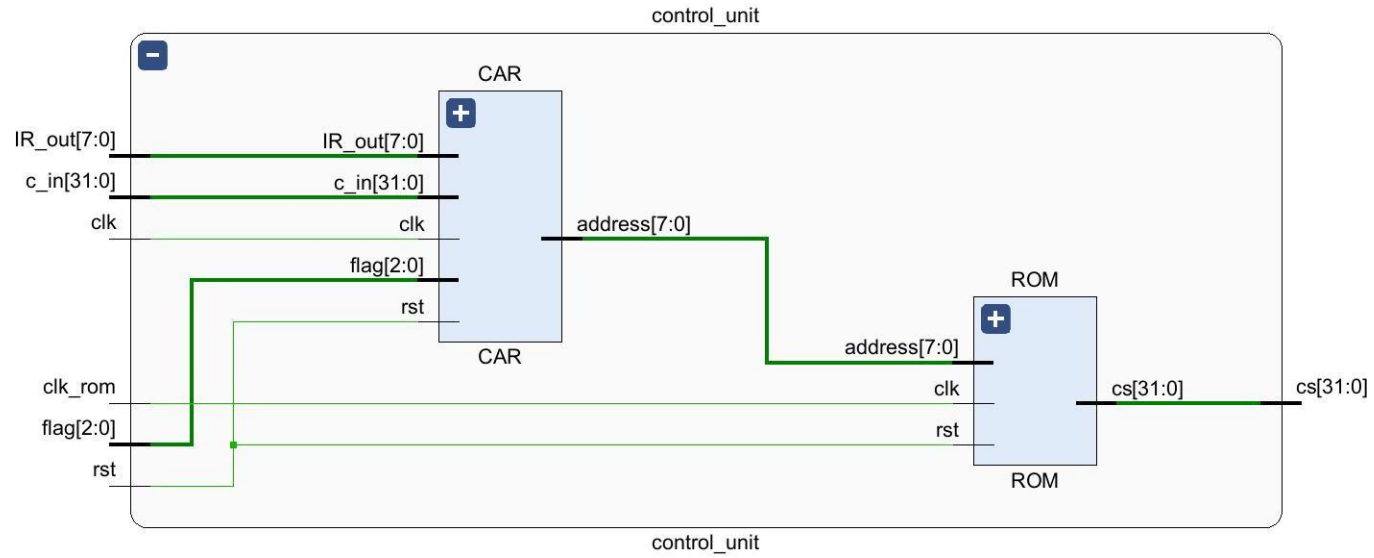
AND(0A)	22	MAR<=MBR[7:0],PC<=PC+1,CAR<=CAR+1	C5，C6，C0
	23	MBR<=memory,CAR<=CAR+1	C3, C0
	24	BR<=MBR,CAR<=CAR+1	C7,C0
	25	ALU<=BR,ALU<=ACC,CAR<=CAR+1	C15,C14,C0
	26	ALU<=ACC and BR,CAR<=CAR+1	C22,C0
	27	ACC<=ALU,CAR<=0	C16,C2
OR(0B)	28	MAR<=MBR[7:0],PC<=PC+1,CAR<=CAR+1	C5，C6，C0
	29	MBR<=memory,CAR<=CAR+1	C3, C0
	30	BR<=MBR,CAR<=CAR+1	C7,C0
	31	ALU<=BR,ALU<=ACC,CAR<=CAR+1	C15,C14,C0
	32	ALU<=ACC or BR,CAR<=CAR+1	C23,C0
	33	ACC<=ALU,CAR<=0	C16,C2
NOT(0C)	34	MAR<=MBR[7:0],PC<=PC+1,CAR<=CAR+1	C5，C6，C0
	35	MBR<=memory,CAR<=CAR+1	C3, C0
	36	ACC<=MBR,CAR<=CAR+1	C8,C0
	37	ALU<=ACC,CAR<=CAR+1	C14,C0
	38	ALU<=not ACC,CAR<=CAR+1	C24,C0
	39	ACC<=ALU,CAR<=0	C16,C2
SHR（0D）	40	ALU<=ACC,CAR<=CAR+1,PC<=PC+1	C14,C0,C6
	41	ALU<=SHR ACC,CAR<=CAR+1	C26,C0
	42	ACC<=ALU,CAR<=0	C16,C2
SHL（0E）	43	ALU<=ACC,CAR<=CAR+1,PC<=PC+1	C14,C0,C6
	44	ALU<=SHL ACC,CAR<=CAR+1	C25,C0
	45	ACC<=ALU,CAR<=0	C16,C2
SAR（10）	46	ALU<=ACC,CAR<=CAR+1,PC<=PC+1	C14,C0,C6
	47	ALU<=SAR ACC,CAR<=CAR+1	C30,C0
	48	ACC<=ALU,CAR<=0	C16,C2
SAL（0F）	49	ALU<=ACC,CAR<=CAR+1,PC<=PC+1	C14,C0,C6
	50	ALU<=SAL ACC,CAR<=CAR+1	C29,C0
	51	ACC<=ALU,CAR<=0	C16,C2
MPY（08）	52	MAR<=MBR[7:0],PC<=PC+1,CAR<=CAR+1	C10，C6，C0
	53	MBR<=memory,CAR<=CAR+1	C3, C0
	54	BR<=MBR,CAR<=CAR+1	C7,C0
	55	ALU<=BR,ALU<=ACC,CAR<=CAR+1	C15,C14,C0
	56	ALU<=ACC * BR,CAR<=CAR+flag(3)	C27,C18
	57	ACC<=ALU,CAR<=0	C16,C2
DIV（09）	58	MAR<=MBR[7:0],PC<=PC+1,CAR<=CAR+1	C10，C6，C0
	59	MBR<=memory,CAR<=CAR+1	C3, C0
	60	BR<=MBR,CAR<=CAR+1	C7,C0
	61	ALU<=BR,ALU<=ACC,CAR<=CAR+1	C15,C14,C0
	62	ALU<=ACC / BR,DR<=ACC%BR,CAR<=CAR+1	C28,C0
	63	ACC<=ALU,CAR<=0	C16,C2
JMP（06）	64	PC<=MBR[7:0],PC=PC+1	C10,C0
JMPGEZ（05）	65	CAR=CAR+1+flag(1) is 0, jump to 2.	C17
	66	CAR<=0,PC<=MBR[7:0]	C10,C2
	67	CAR<=0,PC<=PC+1	C6,C2

HALT（07）	68	CAR<=0	C2

- ROM for control signals Table.

The ROM table of control signals can be formed by storing the above control signal sequence in ROM, please refer to my_rom3.coe file in the attachment for details.

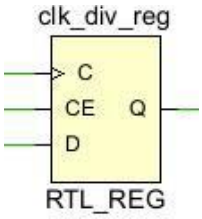
- Control Unit Input/Output Port Diagram



8. crossover modules

The divider is used to delay reading the ROM table by one cycle.

- Input and output port diagrams



9. overall schematic

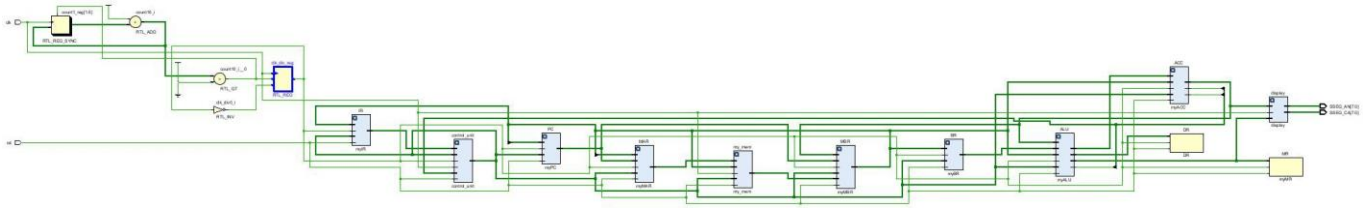


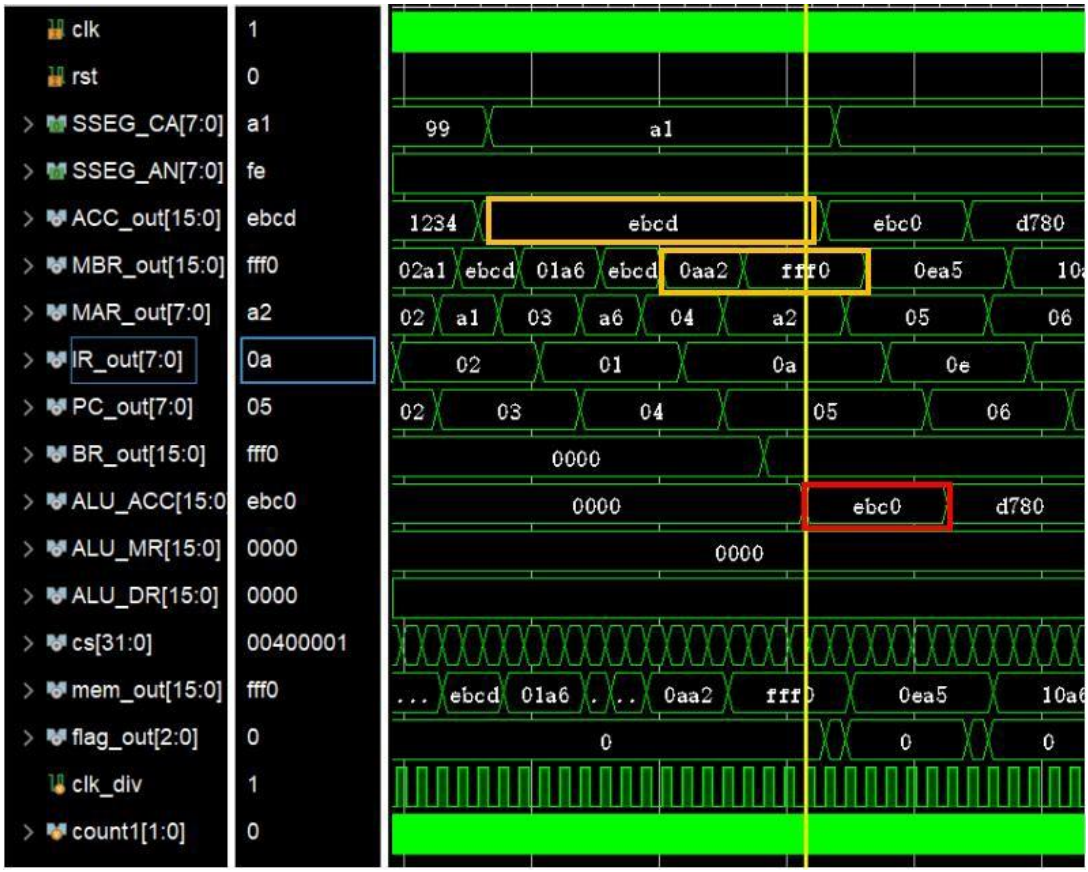
Figure 3

10. Simulation tests

11. command to validate the results

- AND

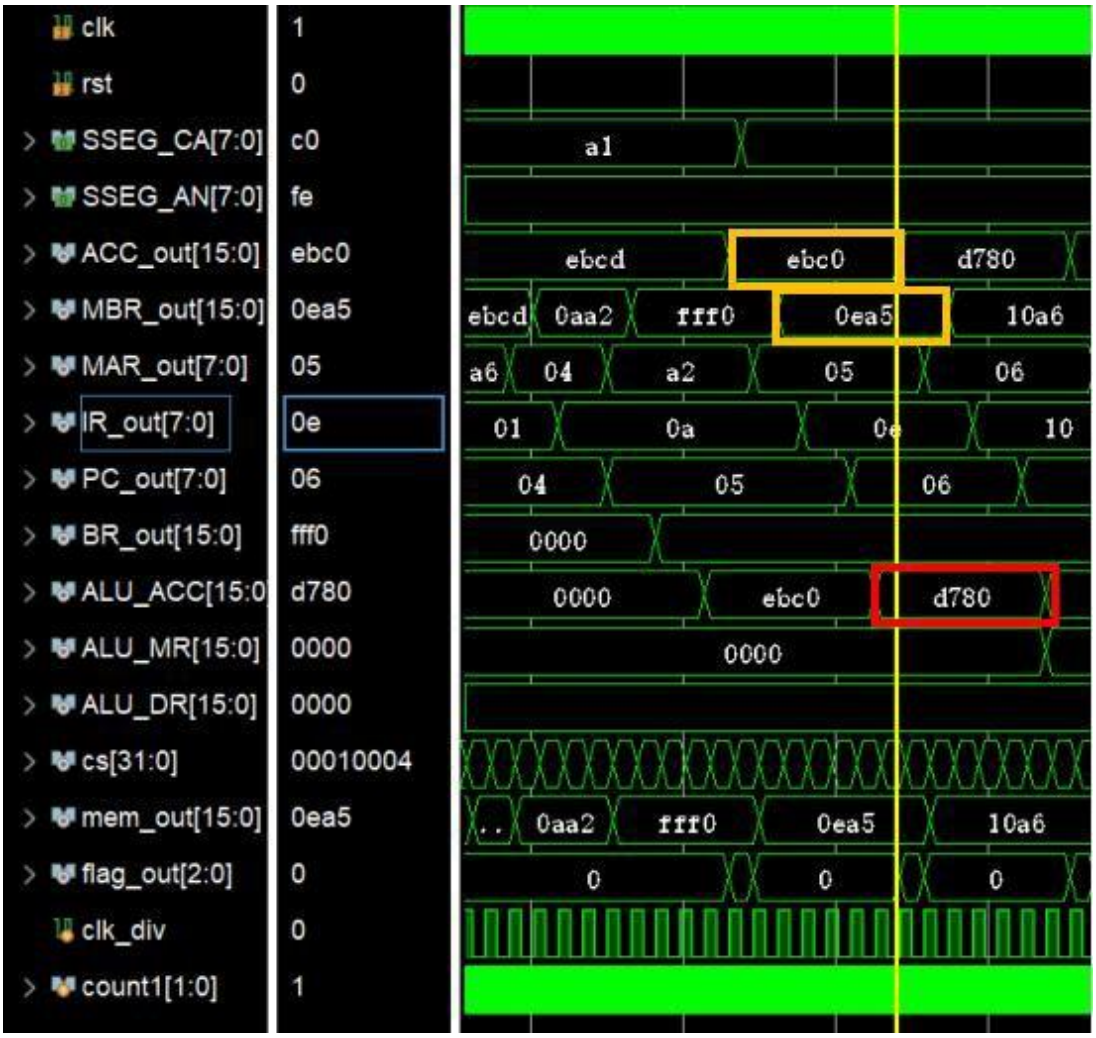
The instruction AND A2 will do a bitwise sum of the value EBCD in ACC and fff0



stored in the address of A2, and the result is ebc0.

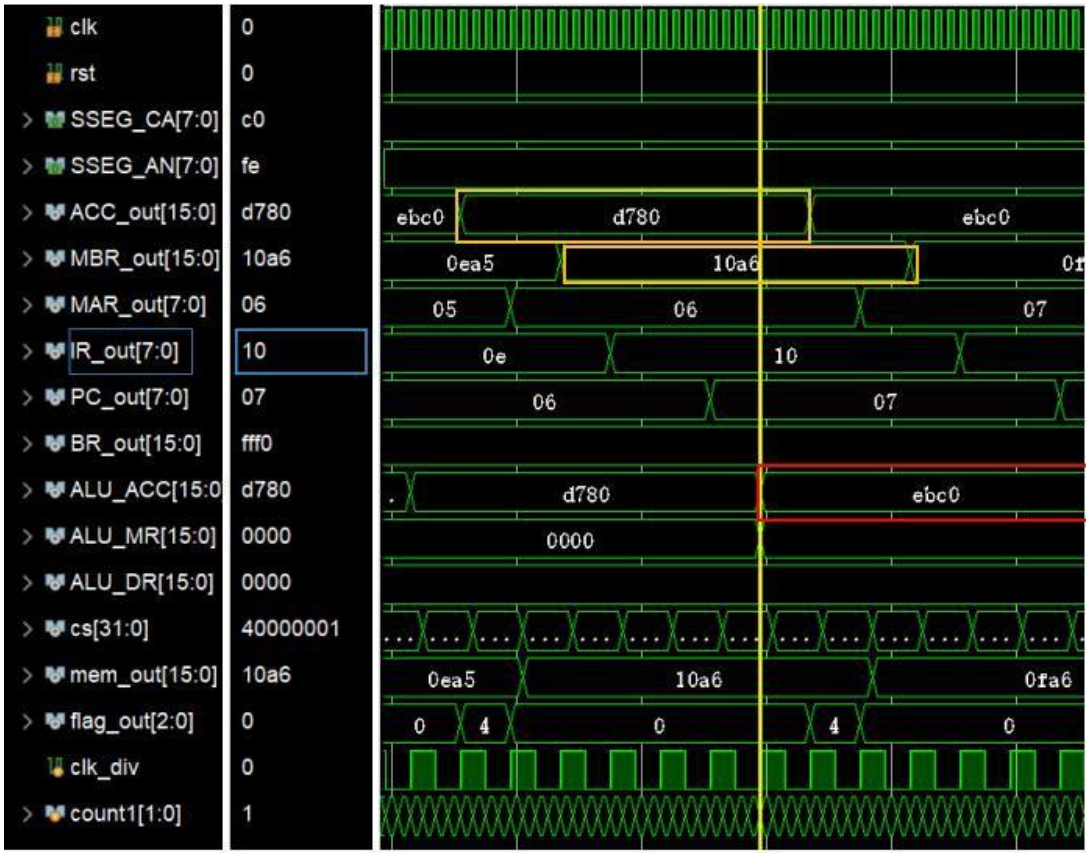
- SHL

The instruction SHL shifts the value EBC0 in ACC by one bit, and the result is D780.



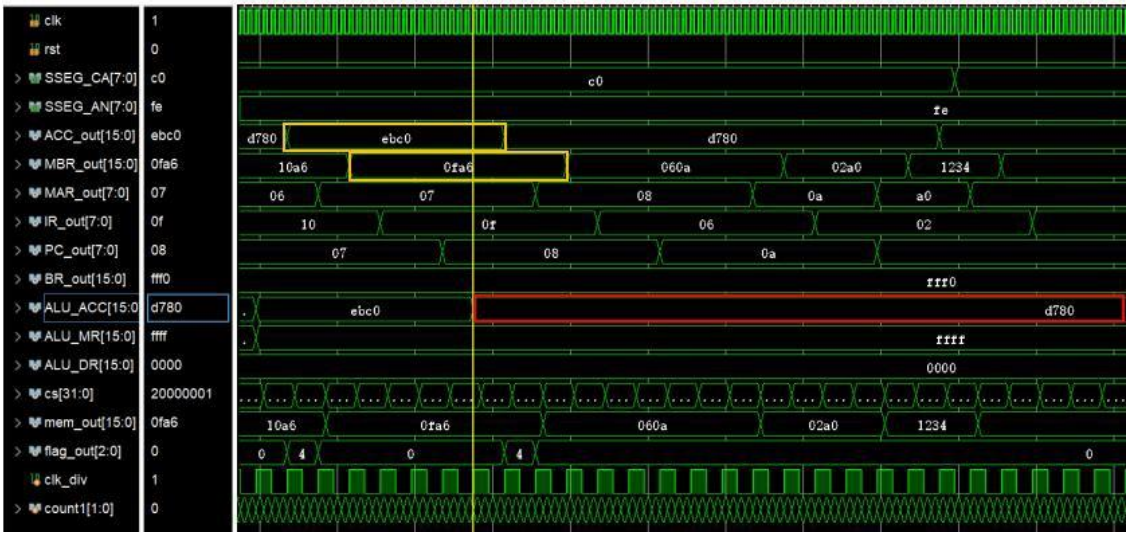
- SAR

The instruction SAR shifts the value D780 in ACC to the right by one bit, D780 is negative, the high bit is complemented by 1, and the result is EBC0.



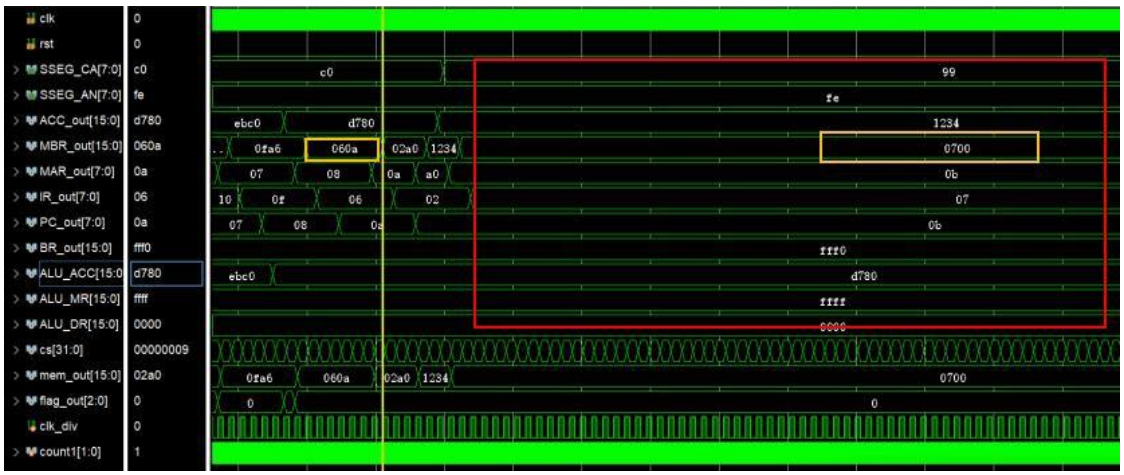
- SAL

The instruction SAL shifts the value EBC0 in ACC by one bit, and the result is D780.



- JMP

The instruction JMP causes the program to jump to the instruction HALT in the address A, all operations are suspended, and the registers are not changed



The remaining operations are checked for correct results in the comprehensive simulation and are not checked separately.

12. integrated simulation

- 1+2+…+100

Actual calculations are 13ba Simulation results.

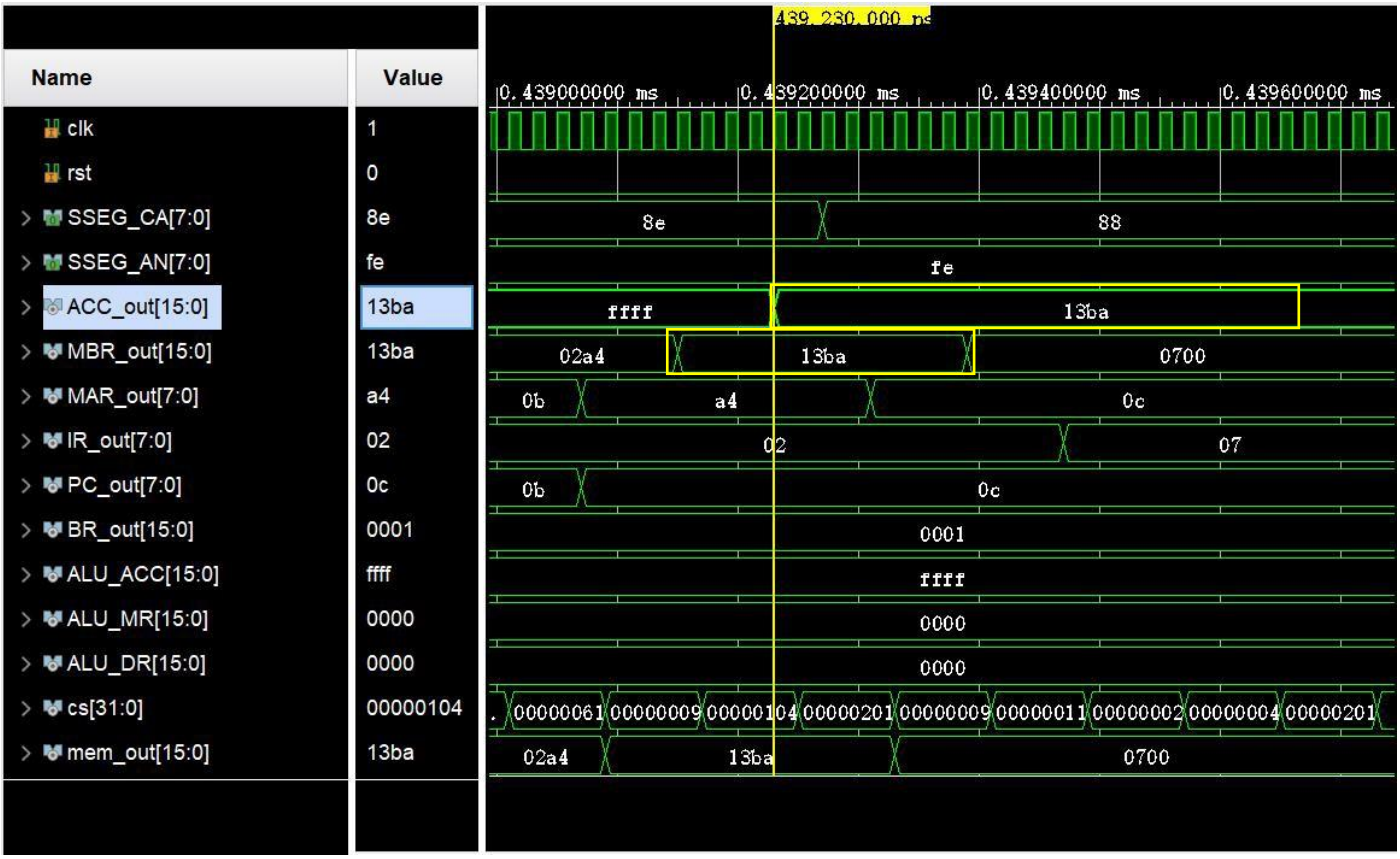
RAM counterpart coe file for concatenation.

1+2+…+100		Address(PC)	content
sum=0	LOAD A0	00	02A0
	STORE A4	01	01A4
temp=100	LOAD A2	02	02A2
	STORE A3	03	01A3
loop:sum=sum+temp	LOOP:LOAD A4	04	02A4
	ADD A3	05	03a3
	STORE A4	06	01a4
temp=temp-1	LOAD A3	07	02a3
	SUB A1	08	04a1

	STORE A3	09	01a3
if temp>=0,goto loop	JMPGEZ	11	0504
	LOAD A4	1 0	02a4
end	HALT	12	0700
		A0	0
		A1	1
		A2	64
		A3	63
		A4	64

Simulation results.

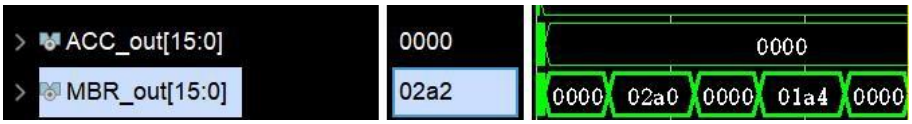
Because of the loop loop used for concatenated addition, it will be negative only when the temp is reduced to less than 0. At this time, ACC_out shows ffff, and it needs to load the result stack again in order to show the correct result of concatenated addition in ACC_out. The final result in the simulation is correctly 13ba.



Implementation of each instruction.

1, sum = 0, the assembly language corresponds to the load address A0 corresponding to the contents of the address A4, does not change the value of the original data. Load the operation code is

02,store opcode is 01

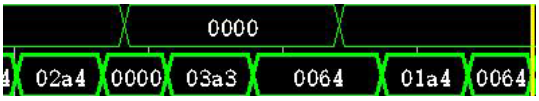


From the above figure, we can see that the value of a4 displayed when the 01a4 instruction is read is 0000.

2, temp = 64, the assembly language corresponds to the value of the load address A2 to address A3, does not change the value of the original data.

As you can see from the above figure, 64 (hexadecimal) has been stored at address a3.

3, sum = sum + temp, load the value of sum, sum for addition, ADD temp location, the result is then deposited into the location of sum, add the operation code for 03.



After one round of arithmetic the sum result is 64.

4, temp = temp-1, and step 3 is similar, you need to first load the value of temp, and then -1 and then deposited into the location of temp. sub opcode is 04.

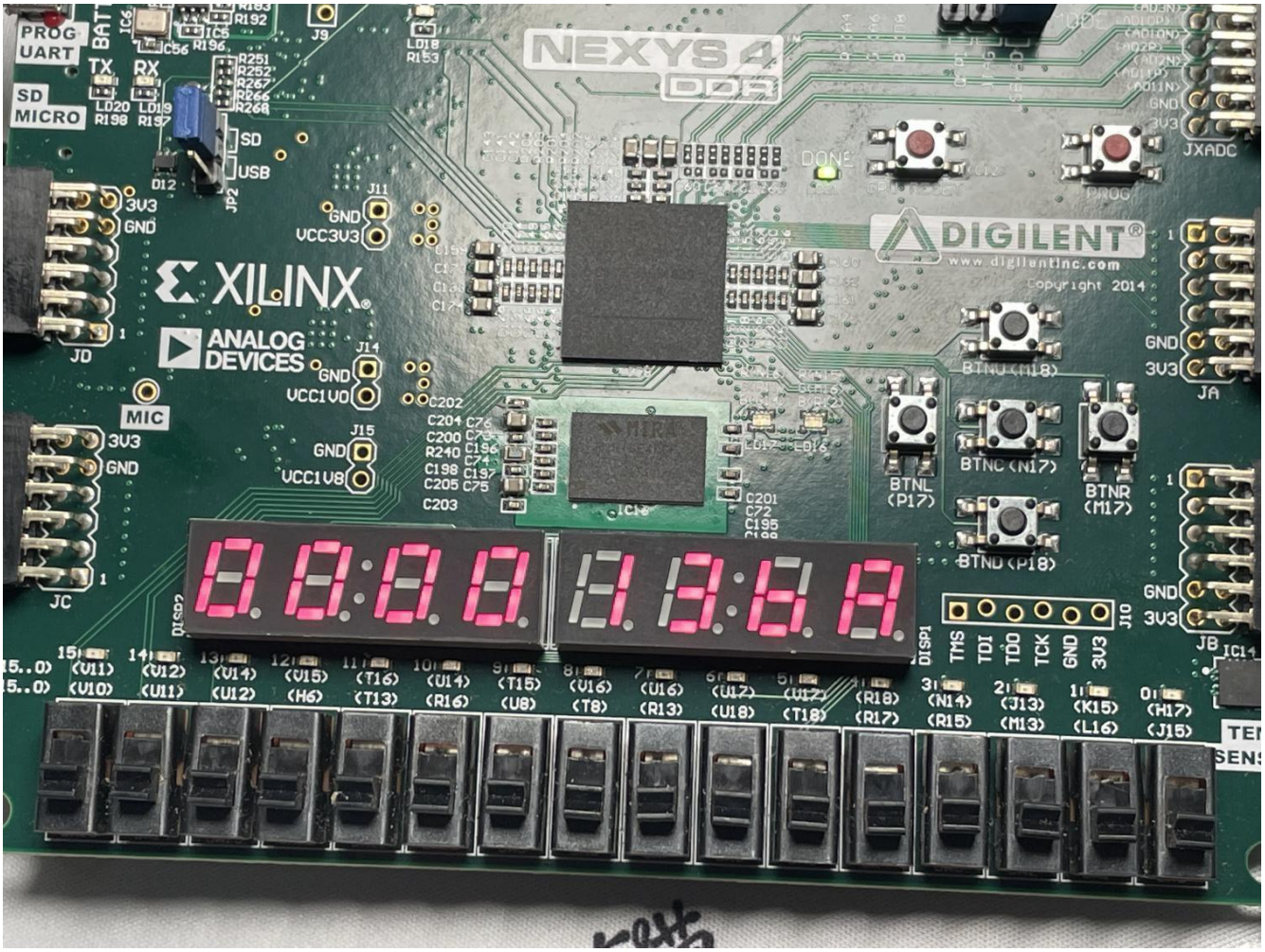
You can see that the value of the last temp address has changed to 0063.

5, JMPGEZ, used to determine whether the temp is reduced to a negative number, the instruction opcode 05

Temp is not negative, go back to sum=sum+temp

and loop there to get the final sum

Board Status.



The display is working well as it can show the correct result after being mounted on the board.

- 31245 * (not 17956) SHR 4 or 12986

The coe table of the RAM corresponding to the synthesized arithmetic.

n1=7a0d	LOAD A0	02a0	00
	STORE A5	01a5	01
n2=4624	LOAD A1	02a1	02
	STORE A6	01a6	03
not n2	NOT A6	0Ca6	04
result=n1*(not n2)	MPY A5	08a5	05
shr result	SHR A6	0DA6	06
shr result	SHR A6	0DA6	07
shr result	SHR A6	0DA6	08
shr result	SHR A6	0DA6	09
result or a2	OR A2	0Ba2	10
end	HALT	0700	11
		7A0D	A0
		4624	A1
		32BA	A2
		3	A3
		1	A4

Theoretical results: according to the calculator to get the result is DE8E 3EFB, need to pay attention to is, in the calculator although it is complementary code but if there is no high bit, the two low bit multiplication is not to determine the positive and negative. Therefore, we need to set the corresponding hexadecimal number to DWORD before the multiplication, and then set it to WORD after the multiplication is finished, after which the value of the high bit will not be changed.

Fig. 1 Multiplication operation

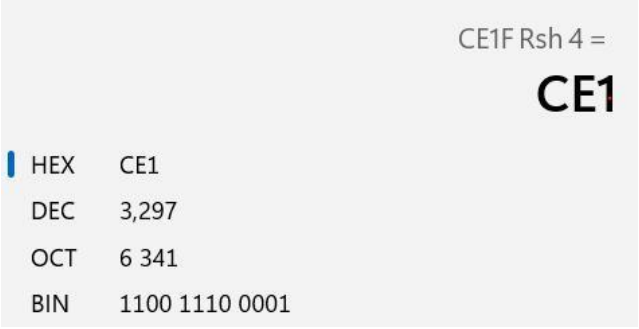


Figure 2 Logic shifted 4 bits to the right

Fig. 3 The or operation

Simulation results.

The final board up gets 3efbde8e, too

13. Questions and Reflections

- the issue of the system partitioning module

  For the design of complex systems, module partitioning is particularly important, and a good command of module partitioning skills can greatly reduce the difficulty of the entire system design.
- complementary arithmetic problems

When designing the ALU module, we used the knowledge of complementary operation and realized the convenience of complementary operation, which can convert subtraction to addition without worrying that the result of multiplication will not be displayed correctly in the lower bit if the result of multiplication is not overflowed by putting the result of multiplication in two registers.

Kwok Chung's personal summary.

In this experiment, I chose a centralized time period to complete the whole module writing, debugging and testing work. Although the process is very hard, but the result is very fruitful.

First of all, I have a better understanding of Vivado programming. In the POC (Proof of Concept) stage, I could only write each module through the state machine method. However, in the CPU design phase, I gradually understood that I should first define the inputs and outputs of each module, and that except for the ALU (Arithmetic Logic Unit), the other modules do not need to perform any operation, but only need to control the reading and writing of the internal registers under the operation of control signals, and the ALU part has an additional step of register operation under the control of control signals. For the naming of module inputs and outputs, I think the best way is "module_in" and "module_out", which is simple and easy to understand, and also reduces the errors in the top-level module routines.

Secondly, my debugging ability has been improved. Since I learned many solutions to errors in advance, I was able to quickly locate and solve the same problems in the exam. In addition, I could not get the correct answer at once during the simulation process, but by analyzing the error results, I was able to propose a solution on my own and predict its feasibility.

Finally, my mindset has improved in the face of difficulties, and I have gradually become more resilient, more confident in myself, and able to solve problems properly no matter what they are. I think this experiment is a thorough growth experience, and I hope that the experience and methods summarized in this experiment can help me in my future experiments and work.

14. Group division of labor.

KWOK LING	Wang Mengjun
Responsible for MAR, PC, IR, CU, TOP documentation	Responsible for MBR, ALU, BR, ACC writing
Responsible for writing control signals, control signal bits, opcodes, and	Responsible for writing displays
Responsible for debugging and simulating each part individually	responsible for the simulation of some of the instructions
Responsible for the simulation of synthesized commands, the	Responsible for the simulation of synthesized commands, the
Responsible for the completion of reports	Responsible for the completion of reports
Percentage of work: 60%	Percentage of work 40%

Appendix

- **ControlUnit.v**

```
`timescale 1ns / 1ps module
control_unit( input clk,
input clk_rom,
input rst,
input [31:0]c_in,//control signal
input [7:0]IR_out,//external input opcode
input [2:0] flag,//0 is positive or negative 1 is multiplication complete 2 is run complete
output [31:0]cs///Output to other modules after control signal is read by rom
    );
wire [7:0]address;//address in rom table
CAR CAR(
.clk(clk),
.rst(rst),
```

```
.IR_out(IR_out),
.c_in(c_in),
.flag(flag),
.address(address)
);
ROM ROM(
.clk(clk_rom),
.rst(rst),
.address(address),
.cs(cs)
);
endmodule
```

- **CAR**

```
`timescale 1ns / 1ps
module CAR(//CAR is to provide the address of the ROM micro-operation according to the opcode of the IR
input clk,rst, input
[7:0]IR_out, input
[31:0]c_in,
input [2:0] flag,//flag0 is positive or
negative,flag1 is end of multiplication,flag2 is
end of processing output [7:0]address
      );
reg [7:0] CAR;
always@(posedge clk)
begin
if(rst==1) CAR<=8'd0;
  else if (rst==0)begin
  if(c_in[2]==1) CAR<=0;
  if(c_in[17]==1)
  CAR<=CAR+1+flag[0];
  if(c_in[18]==1)
  CAR<=CAR+flag[1];//wait for
  multiplication run to finish
  if(flag[2]==1)//if processing is complete
  if(c_in[0]==1)
  CAR goes to 0 CAR<=0;
  CAR<=CAR+1;
  if(c_in[1]==1)begin //decoding is done
  directly here case(IR_out)//instruction
  opcode 8'h01:CAR=8'd7;//STORE
  8'h02:CAR=8'd4;//LOAD
  8'h03:CAR<=8'd10;//ADD
  8'h04:CAR<=8'd16;//SUB
  8'h05:CAR<=8'd65;//JMPGEZ
  8'h06:CAR<=8'd64;//JMP
  8'h07:CAR<=8'd68;//HALT
  8'h08:CAR<=8'd52;//MPY
  8'h09:CAR<=8'd58;//DIV
```

```
  8'h0A:CAR<=8'd22;//AND
  8'h0B:CAR<=8'd28;//OR
  8'h0C:CAR<=8'd34;//NOT
  8'h0D:CAR<=8'd40;//SHR
  8'h0E:CAR<=8'd43;//SHL
  8'h0F:CAR<=8'd49;//SAL
  8'h10:CAR<=8'd46;//SAR
default
CAR<=8'd0;//address
endcase
    end
end
end
assign address = CAR;
endmodule
```

- **ROM**

```
`timescale 1ns / 1ps
module ROM(
input clk,
input rst,
input [7:0] address,
output [31:0]cs
        );
my_rom my_rom (
    .clka(clk),          // input wire clka
    .ena(~rst),
    .addra(address),      // input wire [7 : 0] addra
    .douta(cs)     // output wire [31 : 0] douta
);
endmodule
```

- **ACC.v**

```
`timescale 1ns / 1ps
module
myACC( input clk,rst,
input [15:0] MBR_out,
input [15:0] ALU_out,
input [31:0]cs,
output [15:0] ACC_out,
output flag0,flag2/0/0 represents positive or negative, 2 represents the end of processing
);
reg [15:0] ACC;
reg flag0_reg;
reg flag2_reg;
always@(posedge clk)
begin
if(rst==1)begin ACC<=0;
  flag0_reg<=0;
```

```
 flag2_reg<=0;end
 else if(rst==0)begin
 if(cs[16]==1)
  begin
       ACC<=ALU_out;
       flag2_reg<=1;//ALU output the result to ACC,
       operation completed end
  else
       begin
       flag2_reg<=0;
       end
 if(cs[12]==1)
 ACC<=0;
 if(cs[8]==1)
       ACC<=MBR_out;
 if(ACC[15]==0)
         flag0_reg<=0;
      else
         flag0_reg<=1;
  end
end
assign ACC_out = ACC;
assign flag0=flag0_reg;
assign flag2=flag2_reg;
endmodule
```

- **ALU.v**

```
`timescale 1ns / 1ps
module
myALU( input
clk,rst,
input wire signed [15:0]    BR_out,
input wire signed [15:0]    ACC_out,
input [31:0] cs,
input wire signed [15:0] MBR_out,
output signed [15:0] ALU_ACC,
output signed [15:0]ALU_MR,//used in
multiplication operation output signed
[15:0]ALU_DR,
output flag1// Judge whether the multiplication is complete or not
);
reg signed [31:0] ALU_out;//store the
result reg signed [15:0] BR_reg.
reg signed [15:0] ACC_reg;
reg signed [15:0] DR_reg;
reg flag1_reg;
always@(posedge clk)
begin
if(rst==1)
begin
ALU_out<=32'd0;
```

```
BR_reg<=16'd0;
ACC_reg<=16'd0;
DR_reg<=0;
flag1_reg<=0;
end
  else if(rst==0)
begin
if(cs[14]==1)
    ACC_reg<=ACC_out;
if(cs[15]==1)
    BR_reg<=BR_out;
if(cs[20]==1)
    ALU_out[15:0]<=ACC_reg+BR_reg;
if(cs[21]==1)
    ALU_out[15:0]<=ACC_reg-BR_reg;
if(cs[22]==1)
    ALU_out[15:0]<=ACC_reg&BR_reg;
if(cs[23]==1)
    ALU_out[15:0]<=ACC_reg|BR_reg;
if(cs[24]==1)
    ALU_out[15:0]<=~ACC_reg;
if(cs[25]==1)
    ALU_out[15:0]<=ACC_reg<<1;
if(cs[26]==1)
    ALU_out[15:0]<=ACC_reg>>1;
if(cs[27]==1)
    ALU_out<=ACC_reg*BR_reg;
    flag1_reg<=1;
if(cs[28]==1)begin
      ALU_out<=ACC_reg/BR_reg;
      DR_reg<=ACC_reg%BR_reg;end
if(cs[29]==1)
      ALU_out[15:0]<=ACC_reg<<<1;
if(cs[30]==1)
      ALU_out[15:0]<=ACC_reg>>>1;end
end
assign ALU_ACC = ALU_out[15:0];
assign ALU_MR = ALU_out[31:16];
assign ALU_DR = DR_reg;
assign flag1=flag1_reg;
endmodule
```

- **BR.v**

```
`timescale 1ns / 1ps
module
myBR( input clk,rst,
input [15:0] MBR_out,
input [31:0]cs,
output [15:0] BR_out
```

```
);
reg [15:0] BR;
always@(posedge clk)
begin
if(rst==1)
  BR<=0;
  else if(rst==0)
    if(cs[7]==1)
    BR<=MBR_out;
end
assign BR_out =BR;
endmodule
```

- **IR.v**

```
`timescale 1ns / 1ps
module myIR(
input clk,rst,
input [15:0] MBR_out,
input [31:0]cs,
output [7:0] IR_out
);
reg [7:0] IR;
always@(posedge clk)
begin
if(rst==1)
  IR<=0;
  else if(rst==0)begin
  if(cs[4]==1)
  IR<=MBR_out[15:8];
    end
end
assign IR_out = IR;
endmodule
```

- **MAR.v**

```
`timescale 1ns / 1ps
module
myMAR( input clk,rst,
input [31:0]cs,
input [7:0] MBR_out,
input [7:0] PC_out,
output [7:0] MAR_out
       );
reg [7:0] MAR;
always@(posedge clk)
if(rst==1)
    MAR<=8'd0;
    else if(cs[5]==1)
      MAR<=MBR_out[7:0];//Low bit to MAR for direct addressing
        else if(cs[9]==1)
```

```
        MAR<=PC_out;//fetch
instruction assign MAR_out =
MAR; endmodule
```

- **MBR.v**

```
`timescale 1ns / 1ps
module myMBR(
input clk,rst, //clock and reset signals
input [15:0] mem_out, //external memory
written to MBR input [7:0] PC_out, //PC
instruction address written to MBR
input [15:0] ACC_out，//ACC运算结果写入MBR
输入[31:0]cs，//16位控制信号
output [15:0] MBR_out
);
reg [15:0] MBR;//MBR Memory is used to hold the value to be stored in the mem and the latest value read
from the mem
always@(posedge clk)
begin
if(rst==1)//reset
   MBR<=16'd0;
   else if(rst==0)
     if(cs[3]==1)
          MBR<=mem_out; else
               if(cs[13]==1)
                   MBR<=ACC_out;
 end
assign MBR_out= MBR;
endmodule
```

- **PC.v**

```
`timescale 1ns / 1ps
module myPC( input
clk,rst,
input [7:0] MBR_out,
input [31:0]cs, output
[7:0] PC_out
);
reg [7:0] PC;
always@(posedge clk)
begin
if(rst==1)
  PC<=0;
  else if(rst==0)begin
    if(cs[10]==1)
    PC<=MBR_out[7:0];
    else if(cs[6]==1)
    PC<=PC+1;end
end
assign PC_out = PC;
endmodule
```

- **Top.v**

```
`timescale 1ns / 1ps
module
top_mod( input
clk,rst,
output [7:0] SSEG_CA,
output [7:0] SSEG_AN
        );
wire [15:0]ACC_out;
wire [15:0]MBR_out;
wire [7:0]MAR_out;
wire [7:0]IR_out;
wire [7:0]PC_out;
wire [15:0]BR_out;
wire [15:0]ALU_ACC;
wire [15:0]ALU_MR;
wire [15:0]ALU_DR;
wire [31:0]cs;
wire [15:0]mem_out;
wire [2:0] flag_out;
reg clk_div=0;
reg [1:0] count1=0;
//Crossover
always @(posedge clk) begin
  count1=count1+1;
  if(count1>1)begin
  count1=0;
  clk_div=~clk_div;
  end
end
display display(
        .clk(clk),
        .acc_out(ACC_out),
        .mr_out(ALU_MR),
        .sseg_an(SSEG_AN),// enable terminal
        .sseg_ca(SSEG_CA)//digital pin
        );
my_mem my_mem(
.clk(clk),
.rst(rst),
.cs(cs),
.MAR_out(MAR_out),
.MBR_out(MBR_out),
.mem_out(mem_out)
);
myMAR MAR(
.clk(clk_div),
.rst(rst),
.MBR_out(MBR_out),
.PC_out(PC_out),
```

```
.MAR_out(MAR_out),
.cs(cs)
);
myMBR MBR(
.clk(clk_div),
.rst(rst),
.mem_out(mem_out),
.PC_out(PC_out),
.ACC_out(ACC_out),
.MBR_out(MBR_out),
.cs(cs)
);
myPC PC(
.clk(clk_div),
.rst(rst),
.MBR_out(MBR_out),
.cs(cs),
.PC_out(PC_out)
);
myIR IR(
.MBR_out(MBR_out),
.IR_out(IR_out),
.cs(cs),
.clk(clk_div),
.rst(rst)
);
myBR BR(
.clk(clk_div),
.rst(rst),
.MBR_out(MBR_out),
.cs(cs),
.BR_out(BR_out)
);
myACC ACC(
.clk(clk_div),
.rst(rst),
.MBR_out(MBR_out),
.ALU_out(ALU_ACC),
.cs(cs),
.ACC_out(ACC_out),
.flag0(flag_out[0]),
.flag2(flag_out[2])
);
myALU ALU(
.clk(clk_div),
.rst(rst),
.BR_out(BR_out),
.ACC_out(ACC_out),
```

```
.cs(cs),
.MBR_out(MBR_out),
.ALU_ACC(ALU_ACC),
.ALU_MR(ALU_MR),
.flag1(flag_out[1]),
.ALU_DR(ALU_DR)
);
myMR MR(
.clk(clk_div),
.rst(rst),
.ALU_MR(ALU_MR)
);
DR DR(
.clk(clk_div),
.rst(rst),
.ALU_DR(ALU_DR)
);
control_unit control_unit(
.clk(clk_div),// difference of one clk?
.clk_rom(clk),
.rst(rst),
.c_in(cs),
.IR_out(IR_out),
.flag(flag_out),
.cs(cs)
);

Endmodule
```

- **Memory（RAM）**

```
`timescale 1ns / 1ps
module
my_mem( input
clk,rst,
input [31:0]cs,
input [7:0]MAR_out,//input PC address
input [15:0]MBR_out,
output [15:0] mem_out
      );
ram RAM(
   .clka(clk),        // input wire clka
   .wea(cs[11]),          // input wire [0 : 0] wea
   .addra(MAR_out),    // input wire [7 : 0] addra
   .dina(MBR_out),       // input wire [15 : 0] dina
   .douta(mem_out)    // output wire [15 : 0] douta
);
endmodule
```

- **Display**

```
module display(
```

```
    input clk,
    input [15:0]acc_out,
    input [15:0]mr_out,
    output reg[7:0] sseg_an,//enable
    terminal output reg[7:0]
    sseg_ca//digital pin
);

    reg[18:0] regN = 0;
    reg[3:0] hex_in = 0;// segment selection
    control signal always @(posedge clk)
    regN <= regN+1 ;

    always @(posedge clk)
    begin
        case (regN[18:16])
            3'b000:
            begin
                sseg_an<= 8'b11111110;
                hex_in<=acc_out[3:0];
            end
            3'b001:
            begin
                sseg_an<= 8'b11111101;
                hex_in<=acc_out[7:4];
            end
            3'b010:
            begin
                sseg_an<= 8'b11111011;
                hex_in<=acc_out[11:8];
            end
            3'b011:
            begin
                sseg_an<= 8'b11110111;
                hex_in<=acc_out[15:12];
            end
            3'b100:
            begin
                sseg_an<= 8'b11101111;
                hex_in<=mr_out[3:0];
            end
            3'b101:
            begin
                sseg_an<= 8'b11011111;
                hex_in<=mr_out[7:4];
            end
            3'b110:
            begin
                sseg_an<= 8'b10111111;
                hex_in<=mr_out[11:8];
```

```
                end
                default:
                begin
                    sseg_an<=8'b01111111;
                    hex_in<=mr_out[15:12];
                end
            endcase
        end

        always @(posedge clk)
        begin
            case (hex_in)
                        4'h0:
                sseg_ca[7:0]<=8'b11000000; 4'h1:
                sseg_ca[7:0]<=8'b11111001; 4'h2:
                sseg_ca[7:0]<=8'b10100100; 4'h3:
                sseg_ca[7:0]<=8'b10110000; 4'h4:
                sseg_ca[7:0]<=8'b10011001; 4'h5:
                sseg_ca[7:0]<=8'b10010010; 4'h6:
                sseg_ca[7:0]<=8'b10000010; 4'h7:
                sseg_ca[7:0]<=8'b11111000; 4'h8:
                sseg_ca[7:0]<=8'b10000000; 4'h9:
                sseg_ca[7:0]<=8'b10010000; 4'ha:
                sseg_ca[7:0]<=8'b10001000; 4'hb:
                sseg_ca[7:0]<=8'b10000011;  4'hc:
                sseg_ca[7:0]<=8'b11000110; 4'hd:
                sseg_ca[7:0]<=8'b10100001; 4'he:
                sseg_ca[7:0]<=8'b10000110;
                4'hf: sseg_ca[7:0]<=8'b10001110;
            endcase
        end
endmodule
```

- **tb_myTop.v**

```
module sim();
        reg clk = 0;
        reg rst = 1;
        wire [7:0]SSEG_AN;
        wire [7:0]SSEG_CA;

      top_mod top_mod(
        .clk(clk),
        .rst(rst),
        .SSEG_AN(SSEG_AN),//digital pipe enable terminal
        .SSEG_CA(SSEG_CA)//digital pin
        );
        initial
        begin
        clk<=0;
        rst<=1;
```

```
    #100 rst<=0;
     end
    always #10 clk=~clk;
Endmodule
```

Simulation of each module individually (port numbers changed after verification of each module, subsequent tb files are for reference only).

- **tb_PC.v**

```
`timescale 1ns / 1ps
module  tb_myPC();
reg clk,rst;
reg C6,C10;
wire [7:0] PC_MAR;
wire [7:0] PC_MBR;
reg [7:0] MBR_PC;
myPC myPC(
.clk(clk),
.rst(rst),
.C6(C6),
.C10(C10),
.PC_MAR(PC_MAR),
.PC_MBR(PC_MBR),
.MBR_PC(MBR_PC)
);
initial begin
clk<=0;
rst<=1;
MBR_PC<=8'h21;
C6<=0;
C10<=0;
#100 rst<=0;
#100 C6=1;//MAR<=PC
#100 C6=0;
#100 C10=1;//MAR<=PC
#100 C10=0;
end
always #10 clk=~clk;
endmodule
```

- **tb_myAC.v**

```
`timescale 1ns / 1ps
module tb_myACC();
reg clk,rst;
reg C8,C16;
wire [15:0] ACC_ALU;
wire [15:0] ACC_MBR;
wire flag0,flag2;
reg [15:0] MBR_ACC;
reg [15:0] ALU_ACC;
myACC myACC(
```

```
.clk(clk),
.rst(rst),
.C8(C8),
.C16(C16),
.flag0(flag0),
.flag2(flag2),
.ACC_ALU(ACC_ALU),
.ACC_MBR(ACC_MBR),
.MBR_ACC(MBR_ACC),
.ALU_ACC(ALU_ACC)
);
initial begin
clk<=0;
rst<=1;
MBR_ACC<=16'hA132;
ALU_ACC<=16'h2345;
C8<=0;
C16<=0;
#100 rst<=0;
#100 C8=1;//ACC<=MBR, at this time the ACC is negative, the program run is over
#100 C8=0;//
#100 C16=1;//ACC<=ALU, at this time the ACC is positive, the program run is over
end
always #10 clk=~clk;
endmodule
```

- **tb_myALU.v**

```
`timescale 1ns / 1ps
module tb_myALU();
reg clk,rst;
reg  signed[15:0]   BR_ALU;
reg signed[15:0]  ACC_ALU;
reg [31:0] cs;
reg signed[15:0]  MBR_ALU;
wire signed[15:0] ALU_ACC;
wire signed [15:0]ALU_MR;//used in
multiplication operations wire signed
[15:0]ALU_DR.
wire flag1;//whether
multiplication is finished
myALU myALU(
.clk(clk),
.rst(rst),
.cs(cs),
.BR_ALU(BR_ALU),
.ACC_ALU(ACC_ALU),
.MBR_ALU(MBR_ALU),
.ALU_ACC(ALU_ACC),
.ALU_MR(ALU_MR),
.ALU_DR(ALU_DR),
.flag1(flag1)
```

```
);
initial begin
clk<=0;
rst<=1;
BR_ALU<=16'h1234;
ACC_ALU<=16'h1000;
cs<=0;
#100 rst<=0;
#100 cs<=32'h00004041;//ALU<=ACC
#100 cs<=32'd0;
#100 cs<=32'h0000C001;//ALU<=BR,ALU<=ACC
#100 cs<=32'd0;
#100 cs<=32'h00100001;//add
#100 cs<=32'd0;
#100 cs<=32'h00200001;//minus
#100 cs<=32'd0;
#100 cs<=32'h00400001;//and
#100 cs<=32'd0;
#100 cs<=32'h00800001;//or
#100 cs<=32'd0;
#100 cs<=32'h01000001;//not
#100 cs<=32'd0;
#100 cs<=32'h04000001;//SHR
#100 cs<=32'd0;
#100 cs<=32'h02000001;//SHL
#100 cs<=32'd0;
#100 cs<=32'h40000001;//SAR
#100 cs<=32'd0;
#100 cs<=32'h20000001;//SAL
#100 cs<=32'd0;
#100 cs<=32'h08040000;//MPY
#100 cs<=32'd0;
#100 cs<=32'h80000001;//MPY
#100 cs<=32'd0;
#100 cs<=32'h10000001;//DIV
#100 cs<=32'd0;

end
always #10 clk=~clk;
endmodule
```

- **tb_myBR.v**

```
`timescale 1ns / 1ps
module  tb_myBR();
reg clk,rst;
reg C7;
wire [15:0] BR_ALU;
reg [15:0] MBR_BR;
myBR myBR(
```

```
.clk(clk),
.rst(rst),
.C7(C7),
.BR_ALU(BR_ALU),
.MBR_BR(MBR_BR)
);
initial begin
clk<=0;
rst<=1;
MBR_BR<=16'h2341;
C7<=0;
#100 rst<=0;
#100 C7<=1;//MBR<=IR
#100 C7<=0;
end
always #10 clk=~clk;
endmodule
```

- **tb_control_unit.v**

```
`timescale 1ns / 1ps
module tb_control_unit();
reg clk,rst;
reg [31:0]c_in;
reg [7:0]IR_cu;
reg [2:0] flag;
wire [31:0]cs;
control_unit control_unit(
.clk(clk),
.rst(rst),
.c_in(c_in),
.IR_cu(IR_cu),
.flag(flag),
.cs(cs)
);
initial
begin
   clk<=0;
   rst<=1;//rese
   t
   IR_cu<=8'h01;//AND instruction

#100    rst<=0;//fuwei
#100    c_in<=32'h00000441;//CAR<=CAR+1;
#100    c_in<=32'h00000000;
end
always #10 clk=~clk;
endmodule
```

- **tb_myIR.v**

```
`timescale 1ns / 1ps
module tb_myIR();
```

```
reg clk,rst;
reg C4;
wire [7:0] IR_cu;
reg [7:0] MBR_IR;
myIR myIR(
.clk(clk),
.rst(rst),
.C4(C4),
.IR_cu(IR_cu),
.MBR_IR(MBR_IR)
);
initial begin
clk<=0;
rst<=1;
MBR_IR<=8'h21;
C4<=0;
#100 rst<=0;
#100 C4<=1;//MBR<=IR
#100 C4<=0;
end
always #10 clk=~clk;
endmodule
```

- **tb_myMAR.v**

```
`timescale 1ns / 1ps
module tb_myMAR();
reg clk,rst;
reg C5,C9;
reg [7:0] PC_MAR;
reg [7:0] MBR_MAR;
wire [7:0] MAR_mem;
myMAR myMAR(
.clk(clk),
.rst(rst),
.C5(C5),
.C9(C9),
.PC_MAR(PC_MAR),
.MBR_MAR(MBR_MAR),
.MAR_mem(MAR_mem)
);
initial begin
clk<=0;
rst<=1;
PC_MAR<=8'h78;
MBR_MAR<=8'h1A;
C5=0;
C9=0;
#100 rst<=0;
```

```
#100 C5=1;//MAR<=PC
#100 C5=0;
#100 C9=1;//MAR<=PC
#100 C9=0;//MAR<=MBR[15:8]
end
always #10 clk=~clk;
endmodule
```

- **tb_myMBR.v**

```
`timescale 1ns / 1ps
module tb_myMBR();
reg clk;
reg rst;
reg C3,C12,C13;
reg [7:0]PC_MBR;
reg [15:0] ACC_MBR;
reg [15:0] mem_MBR;
wire [15:0]MBR_mem;
wire [7:0] MBR_MAR;//operand address
wire [7:0] MBR_IR;// opcode
wire [15:0] MBR_ACC;//operand
wire [15:0] MBR_BR;//operand
wire
[15:0]MBR_ALU;//operand
wire [7:0] MBR_PC;//new
address myMBR myMBR(
.clk(clk),
.rst(rst),
.C3(C3),
.C12(C12),
.C13(C13),
.PC_MBR(PC_MBR),
.ACC_MBR(ACC_MBR),
.mem_MBR(mem_MBR),
.MBR_mem(MBR_mem),
.MBR_MAR(MBR_MAR),
.MBR_IR(MBR_IR),
.MBR_ACC(MBR_ACC),
.MBR_ALU(MBR_ALU),
.MBR_PC(MBR_PC),
.MBR_BR(MBR_BR)
);

initial begin
clk<=0;
rst=1;
C3=0;
C12=0;
C13=0;
```

```
mem_MBR<=16'h2A7B;
ACC_MBR<=16'h1234;
PC_MBR<=8'h20;
#100 rst=0;
#100 C3=1;//MBR<=mem
#100 C3=0;//MBR<=ACC
#100 C12=1;//MBR<=PC
#100 C12=0;
#100 C13=1;
#100 C13=0;
end
always      #10 clk <= !clk;
endmodule
```